Chapitre 10

Contrôle de la fiabilité des logiciels à évoluer : un canevas pour le développement incrémental et itératif de logiciels à composants et orientés service

10.1. Introduction

Les systèmes logiciels modernes se distinguent par un besoin d'évolution rapide et une complexité croissante, avec notamment l'apparition de nouveaux domaines d'applications, comme par exemple les logiciels destinés aux périphériques mobiles, c'est-à-dire les assistants mobiles tels que les téléphones ou encore les tablettes. Dans ces domaines, les utilisateurs réclament que les logiciels fournissent toujours plus de fonctionnalités. Ces logiciels ont alors besoin de prendre en compte ces nouvelles exigences demandées, tout en ayant conscience du nombre toujours croissant de périphériques hétérogènes.

Cette problématique d'évolution a été pointée dès 1985 par Lehman qui définit la notion d'évolution du logiciel ainsi : *"Continued satisfaction demands continuing changes. The system will have to be adapted to a changing environment, changing needs, developing concepts and advancing technologies. The application and the system should evolve"* [1] [LEH 85]. Dans cette définition, l'évolution d'un logiciel consiste

---

Chapitre rédigé par Guillaume WAIGNIER et Anne-Françoise LE MEUR et Laurence DUCHIEN.

1. La poursuite de la satisfaction exige des changements continus. Le système devra être adapté à un environnement en mutation, à l'évolution des besoins, à l'élaboration de concepts et aux progrès technologiques. L'application et le système devront évoluer.

à adapter le logiciel afin qu'il respecte les nouvelles exigences des utilisateurs et qu'il suive les modifications de l'environnement et les avancées technologiques. Dans le cadre de ce chapitre, nous considérons que l'évolution correspond à adapter le logiciel en ajoutant de nouvelles fonctionnalités et en supprimant les fonctionnalités obsolètes.

Pour répondre aux demandes d'évolution, les méthodes de développement logiciel ont, elles aussi, évolué dans le temps pour aller vers des cycles de développement courts, incrémentaux et itératifs, tels que SCRUM [BEE 99] ou XP [BEC 99] afin d'être plus réactifs aux besoins d'évolution et garantir un certain degré de fiabilité. En effet, non seulement il faut pouvoir intégrer de nouvelles fonctionnalités rapidement mais il faut aussi s'assurer que l'évolution effectuée ne compromet pas le bon fonctionnement de l'application toute entière, ni ne nuit à sa qualité de service.

De façon connexe aux méthodologies de cycle de développement, et pour répondre à la difficulté de construire des logiciels de plus en plus grands et complexes, l'ingénierie du logiciel propose deux grands principes pour l'élaboration de tels systèmes, à savoir la modularisation et la réutilisation. La modularisation vise à encapsuler et isoler, le plus possible, chaque fonctionnalité dans des briques logicielles différentes. Le logiciel résultant est donc vu comme un assemblage de briques qui fournissent dans leur ensemble toutes les fonctionnalités attendues par les utilisateurs. La réutilisation consiste à autoriser l'utilisation des mêmes briques logicielles pour concevoir différents logiciels. Ce principe permet donc d'augmenter la vitesse de développement du logiciel. Une application de ces deux principes a donné lieu au paradigme d'architecture logicielle [MED 00]. L'architecture spécifie les briques logicielles qui composent l'application, ainsi que les interactions entre ces briques.

Deux grandes familles ont émergé de cette approche : les architectures à composants [SZY 97] et les architectures orientées service [ERL 05]. Dans les deux cas, de nombreux modèles ont été proposés, néanmoins aucun ne permet facilement de faire évoluer une application de façon incrémentale et itérative par un support méthodologique et outillé. De plus, si certaines de ces approches se sont intéressées à fournir des outils pour effectuer des analyses statiques ou dynamiques afin de vérifier des propriétés applicatives qui ont un impact sur la fiabilité de l'application formée de composants/services, de nombreux modèles n'apportent aucun support, et ceux qui offrent des solutions se concentrent uniquement sur un type de propriétés. Pour répondre à ces constats, nous avons développé le canevas de conception et d'évolution CALICO qui offre non seulement un cadre de développement incrémental et itératif pour les applications à base de composants/services et, de plus, fédère un ensemble d'analyses pour augmenter la fiabilité des applications.

Ce chapitre est organisé comme suit. Tout d'abord, dans la section 10.2, nous donnons quelques définitions concernant les architectures logicielles et les modèles à composants ou services existants. Dans la section 10.3, nous faisons le point sur les cycles de développement logiciel généralement utilisés, ainsi que la méthode usuelle

de développement des applications à base de composants/service, dans chaque cas en mettant en perspective les possibilités d'évolution que chacune de ces approches offre. Les aspects de fiabilité liés aux propriétés applicatives sont décrits dans la section 10.4 et la méthode de développement de CALICO ainsi que les analyses statiques et dynamiques qu'il supporte sont présentés dans la section 10.5. Enfin la section 10.6 conclut ce chapitre.

10.2. Architectures logicielles : définitions

Une définition bien connue du terme architecture logicielle est proposée par l'association de standard IEEE : "*Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*" [2] [IEE 00]. Bien qu'il n'y ait pas de définition universelle, il est usuellement accepté que la spécification d'une architecture logicielle décrit au moins les entités contenues dans l'architecture ainsi que les relations entre ces entités. Cette section présente les deux grands styles d'architectures logicielles classiquement rencontrés : les architectures à composants et les architectures orientées service.

10.2.1. *Architectures logicielles à composants*

Dans les architectures à composants, l'entité composable est le composant. Szypersky a donné une définition, communément admise, de la notion de composant : "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" [3] [SZY 97]. Cette définition met en avant trois propriétés fondamentales d'un composant logiciel. Premièrement, un composant est une entité qui décrit explicitement les fonctionnalités qu'il offre par le biais d'interfaces contractualisées, ainsi que ses dépendances, c'est-à-dire les fonctionnalités qu'il requiert. Deuxièmement, un composant est une entité composable. Concrètement cela signifie qu'une application à base de composants est en fait un assemblage de composants. De plus, cette composition doit satisfaire les exigences décrites par les interfaces contractualisées des composants. Troisièmement, un composant est une entité capable d'être déployée sur une plate-forme d'exécution, indépendamment des autres composants.

---

2. L'architecture est définie par des pratiques recommandées comme l'organisation fondamentale d'un système, incarnée par ses composants, leurs relations entre eux et avec l'environnement, ainsi que les principes régissant leur conception et leur évolution.

3. Un composant logiciel est une unité de composition avec des interfaces spécifiées contractuellement et des dépendances de contexte explicites. Un composant logiciel peut être déployé de façon autonome et est soumis à la composition par des tiers.

Les architectures à composants ont donné lieu à une multitude de modèles définis par des consortiums industriels ou des laboratoires académiques. Par exemple, le modèle à composants Fractal [BRU 04] a été élaboré en collaboration avec le laboratoire de recherche Inria et France Télécom R&D. Plus récemment, le modèle à composants OSGi [OSG 07] a été élaboré par l'alliance OSGi. Côté industriel, plusieurs modèles à composants sont largement utilisés tels que EJB [Sun 97] et .Net [LOW 05]. Pour le moment, aucun modèle n'a acquis de prédominance par rapport aux autres. Il existe donc une très grande hétérogénéité de modèles et de plates-formes à composants qui coexistent. Chacun de ces modèles repose sur des concepts communs, à savoir les notions de composant, de port, de connecteur et de configuration [MED 00]. Cependant, ces concepts peuvent avoir des noms différents et chacun de ces modèles apporte son lot de règles à suivre pour assembler les composants entre eux :

– **composant :** le composant est une entité de calcul ou de stockage. Il peut être vu comme une boîte blanche ou noire, selon que le contenu du composant est explicité ou pas. Il possède des ports, appelés interfaces contractualisées, qui sont les points d'accès du composant, c'est-à-dire que le composant interagit avec les autres composants exclusivement au travers de ses ports. Dans la majorité des modèles à composants, les ports peuvent être requis ou fournis, décrivant respectivement les fonctionnalités requises ou fournies par le composant [WAI 07]. D'autres modèles à composants offrent des ports complexes regroupant à la fois des fonctionnalités requises et fournies, comme SafArchie [BAR 05] ou UML2 [OMG 07b]. Un composant peut être primitif ou composite. Un composant primitif est un composant indivisible. Un composant composite est vu comme une boîte blanche dont le contenu est constitué d'un assemblage de composants ;

– **connecteur :** un connecteur modélise les interactions entre les composants. Il peut prendre différentes formes suivant le modèle à composants. Par exemple il peut être un simple lien qui représente une communication par appel de méthode ou par envoi de message. Il peut aussi modéliser une interaction plus complexe, par exemple, le connecteur peut être chargé d'effectuer une répartition de charge ou établir une communication transactionnelle. Un connecteur connecte un ou plusieurs ports requis avec un ou plusieurs ports fournis compatibles. La notion de compatibilité entre les ports est un point crucial. Elle est définie par les règles de composition du modèle à composants utilisé ;

– **configuration :** la configuration [MED 00] est un terme historique utilisé pour nommer l'assemblage de composants correspondant à l'application. La description de l'assemblage est structurelle et repose sur la création de connecteurs entre composants. Dans les modèles à composants hiérarchiques, c'est-à-dire offrant la notion de composant composite, la configuration est souvent confondue avec le composant composite de plus haut niveau hiérarchique. Par exemple, la figure 10.1 représente graphiquement un composant composite `HelloWorld` constitué de deux sous-composants ou composants primitifs. Le client `Client` dispose de deux ports, un port fourni `P2` à gauche et un port requis nommé `P3` à droite. Le port requis `P3` est connecté à l'aide

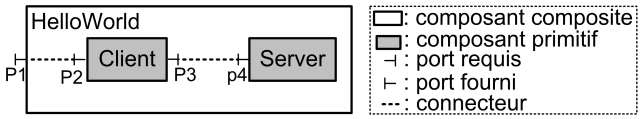


Figure 10.1. *Exemple d'une architecture à composants*

d'un connecteur au port fourni `P4` du composant `Server`.

Un assemblage de composants est cohérent si toutes les règles de composition du modèle à composants sont satisfaites. La cohérence d'une architecture n'inclut donc pas seulement que tous les connecteurs de l'architecture connectent entre eux des composants dits compatibles, c'est-à-dire que les garanties offertes par les composants remplissent les conditions imposées par les composants assemblés, mais prend aussi en compte les règles de composition hiérarchique qu'il faut respecter lors de la création des composants composites. Par exemple, dans la majorité des modèles hiérarchiques, un composant doit être contenu dans exactement un composite.

10.2.2. *Architectures logicielles orientées service*

L'autre grand style d'architectures logicielles correspond aux architectures orientées service. Il existe de nombreuses définitions du terme architecture orientée service [ERL 05] (SOA pour *'Service Oriented Architecture'*) provenant de différents organismes, comme le consortium OASIS [OAS 08] ou le consortium W3C (*World Wide Web Consortium*) [W3C]. Par exemple, la définition donnée par le W3C est : *A service Oriented Architecture is a set of components which can be invoked and whose interface descriptions can be published and discovered* [4].

Cette définition exprime qu'une architecture SOA est la composition d'un ensemble de composants. Ces composants, qui sont par convention appelés des services, exportent les fonctionnalités qu'ils offrent à travers des interfaces. Dans la suite du document, nous nous focalisons sur les services web qui sont une mise en œuvre normalisée du paradigme SOA dédiée au web [ERL 05] :

– service : un service est une action effectuée par un fournisseur de service, comme une entreprise, et produit un résultat final à un consommateur de service. Le service possède donc un point d'accès fourni. Dans le cas des services web, ce point d'accès est appelé un port et doit être spécifié avec le langage standardisé WSDL (*Web Service Definition Language*) [W3C 01]. Il permet de décrire le service, les opérations qu'il

4. Une architecture orientée service est un ensemble de composants qui peuvent être invoquées et dont les descriptions d'interface peuvent être publiées et découvertes

offre et les messages échangés. Le service diffère de la notion de composant. Notamment, un service n'explicite pas ses dépendances, seules les fonctionnalités offertes sont décrites. De plus, le concept de hiérarchisation n'existe pas, c'est-à-dire qu'un service est toujours primitif :

– communication : les services sont faiblement couplés entre eux et communiquent uniquement par échange de messages. La connexion entre les services est dynamique car elle est créée juste avant la communication entre les services et est supprimée à la fin de la communication. Dans le cas des services web, le protocole de communication est SOAP (*Simple Object Access Protocol*). C'est un protocole standardisé reposant sur XML qui décrit les messages échangés entre les services de manière indépendante à tout langage de programmation. Ce protocole favorise ainsi l'interopérabilité entre différents services ;

– composition : la définition de la composition de services repose sur la spécification des interactions entre les services en vue de décrire un processus métier. Cette composition est définie par la description soit d'une orchestration, soit d'une chorégraphie. La spécification de la composition se focalise sur le comportement des services, c'est-à-dire qu'elle exprime l'ordre d'enchaînement des invocations des autres services avec les messages envoyés et reçus. Aucun formalisme n'a été imposé pour spécifier la composition. L'architecte peut utiliser un langage de programmation, comme Java, ou un langage de composition dédié, comme WS-BPEL (*Web Services Business Process Execution Language*) [OAS 07]. Contrairement aux architectures à composants, la composition de service repose sur l'utilisation de services déjà déployés, c'est-à-dire que ces services sont gérés par des organismes tiers et sont accessibles *via* le web. Pour faciliter l'accès à ces services, des services d'annuaires permettent de rechercher dynamiquement le service désiré.

10.2.3. *Architectures hybrides*

Plusieurs modèles à composants et orientés service coexistent. Afin de les réunir et de les faire interopérer, le consortium OpenSOA a défini le modèle industriel SCA (*Service Component Architecture*) [BEA 07]. Ce modèle se concentre sur l'intégration de technologies hétérogènes et leur interopérabilité. Par exemple, cela inclut le support des composants implémentés dans différents langages de programmation, comme Java et C, et différentes technologies, comme les composants OSGi. Différents modes de communication entre composants sont aussi pris en compte, comme Java RMI et WebService. Le modèle SCA est adapté au développement de logiciels qui mettent en relation les entreprises et/ou les particuliers car le support multiprotocoles de communication et multi-implémentations facilite l'interopérabilité.

Nous retrouvons dans le modèle SCA les quatre concepts communs des modèles à composants : le concept de composant, de port, de connecteur et de configuration sous des noms différents. Le modèle à composants SCA prend en charge les composants primitifs et composites. Les ports fournis et requis sont appelés respectivement

services et références. Les connecteurs sont des liens appelés *wire* et la configuration correspond au modèle d'assemblage SCA (*SCA assembly model*).

10.2.4. *Bilan*

Dans tous les modèles, une architecture à composants correspond au moins à une spécification structurelle qui décrit l'assemblage de composants et de connecteurs. Les différences entre les modèles à composants résident dans le mode de communication entre les composants et dans les règles de composition. Concrètement, chaque modèle offre son propre mode de communication, qui peut-être par exemple synchrone par appel de méthodes ou asynchrone par envoi de messages. De même, chaque modèle définit ses propres règles de compatibilité entre les composants et ses règles de composition hiérarchique. Cependant, en dépit de ces différences entre les modèles, la manière de raisonner pour concevoir une architecture est la même. Elle est fondée sur un raisonnement structurel.

Les architectures orientées service reposent sur les concepts de service, de communication et de composition. La composition correspond à une spécification comportementale qui décrit le flot de contrôle, c'est-à-dire l'enchaînement des invocations des autres services. La construction d'une telle architecture nécessite donc de raisonner sur le comportement global du logiciel.

Les architectures à composants et les architectures orientées service reposent sur les mêmes concepts principaux. La différence principale correspond au moyen de définir la composition des entités (composants ou services). Afin de les unifier, de nombreux travaux, comme Acme [MON 01] et SCL [FAB 07], ont comparé les différents modèles à composants et ont aussi fait le rapprochement entre les architectures à composants et les architectures orientées service, comme par exemple [BRE 07].

10.3. Cycle de développement et évolution

Après avoir présenté les architectures logicielles, cette section fait le point sur le processus de développement d'un logiciel. Puis elle décrit trois exemples de cycles de développement et détaille les différentes étapes de développement dans le cas d'une construction d'une architecture logicielle.

10.3.1. *Processus de développement et évolution d'un logiciel*

Le développement d'un logiciel est effectué par une succession d'étapes et de tâches entreprises par des acteurs différents. Le standard ISO 12207 définit toutes les tâches nécessaires pour le développement logiciel [ISO 95]. Ce standard inclut

la description des processus techniques, comme le développement du système et sa maintenance, des processus de support, comme la rédaction de la documentation, et des processus organisationnels.

Le standard ISO 12207 offre un découpage des étapes de très fine granularité, mais globalement l'élaboration du logiciel inclut les cinq grandes étapes suivantes : l'analyse des besoins, la conception, l'implémentation, la validation et le déploiement :

– durant l'étape d'analyse des besoins, un analyste logiciel identifie les besoins du client, c'est-à-dire qu'il établit un cahier des charges fonctionnel et technique qui décrit les fonctionnalités et les contraintes du logiciel et du matériel. Cette étape inclut par exemple l'écriture de scénarios d'utilisation du système ;

– l'étape de conception consiste à décrire le logiciel à un haut niveau d'abstraction, c'est-à-dire sans se soucier des détails d'implémentation. Elle est effectuée par un architecte logiciel :

– lors de l'étape d'implémentation, les développeurs écrivent le code du logiciel ;

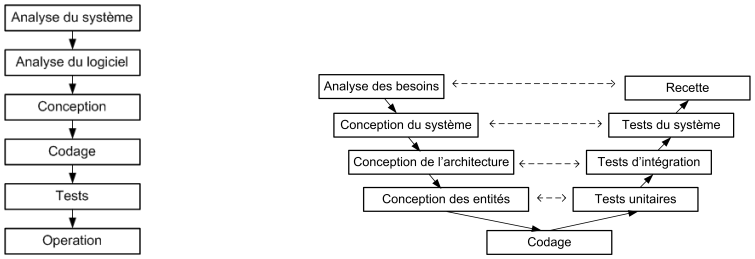
– dans le cas des architectures logicielles, l'étape de validation peut être décomposée en deux étapes distinctes :

- l'étape de validation statique consiste à vérifier que la conception de l'architecture est cohérente, c'est-à-dire qu'elle respecte les exigences ;

- lors de l'étape de validation dynamique, les testeurs exécutent le logiciel afin de s'assurer que le comportement du logiciel et du matériel est conforme à ce qui a été défini lors de l'étape d'analyse des besoins. Le standard ISO ne définit aucun moyen de validation. Par exemple, les testeurs peuvent exécuter les scénarios d'utilisation établis durant l'étape d'analyse des besoins. Cette étape nécessite que le logiciel soit déployé ;

– l'étape de déploiement correspond à l'installation du logiciel sur les différentes machines cibles par un administrateur système.

Le standard ISO définit aussi le processus technique de maintenance (ou d'évolution) du logiciel. Ce processus est activé lorsque le code du système logiciel doit être modifié suite à la découverte d'une erreur ou d'un besoin d'évolution du logiciel. L'objectif est de modifier le système tout en gardant son intégrité. Lorsque le logiciel doit évoluer, le processus technique de développement du système, décrit précédemment, est réexécuté. Le standard ISO 12207 ne spécifie pas d'ordre temporel entre les étapes de développement. Au contraire, il permet la définition de différents enchaînements possibles de ces étapes, appelés des modèles de cycle de développement du logiciel. Par exemple, l'exécution de ces étapes peut être séquentielle, itérative ou se chevaucher.



a) *Cycle en cascade* **b)** *Cycle en V*

Figure 10.2. *Cycle descendant*

10.3.2. *Modèles de cycles de développement*

Dans cette section, nous présentons deux exemples de modèle de cycle descendant, puis nous finissons avec des exemples de modèles de cycle plus réactifs qui facilitent l'évolution du logiciel.

10.3.2.1. *Cycle descendant*

Un premier cycle de développement descendant est le cycle de développement en cascade, appelé modèle *waterfall*. Il a été introduit par Royce en 1987 [ROY 87]. Ce cycle, représenté par la figure 10.2, est constitué de six étapes successives où chaque étape est exécutée si et seulement si l'étape précédente est complètement finie.

Le second cycle de développement largement utilisé, depuis les années 1980, est le cycle de développement en V [IAB 97]. Ce cycle peut être vu comme une extension du cycle de développement en cascade. Néanmoins, au lieu de développer séquentiellement le logiciel, puis les tests, comme cela est fait dans le modèle de cycle en cascade, les tests sont écrits ici en parallèle (voir la figure 10.2). Ce cycle associe donc à chaque étape de conception, une étape de validation. L'objectif du cycle est de limiter, en cas de détection d'anomalies, le retour sur des étapes précédentes.

Les cycles de développement descendants sont linéaires. Ils possèdent un inconvénient majeur. En effet, ils partent de l'hypothèse de pouvoir effectuer une étape uniquement en se basant sur les étapes précédentes. Notamment, ils suggèrent une identification complète des besoins avant de commencer la conception. Or, il n'est pas évident pour un client de pouvoir exprimer parfaitement ses besoins. Généralement, le client a besoin de voir un prototype fonctionnel pour commenter les fonctionnalités qu'il désire. Ces modèles de cycles sont donc rigides et ne sont pas adaptés pour développer des logiciels qui évoluent rapidement.

10.3.2.2. *Cycle de développement itératif et incrémental*

Afin d'introduire plus de souplesse dans le cycle de développement, les modèles de cycle de développement ont évolué et sont devenus itératifs et incrémentaux [LAR 03].

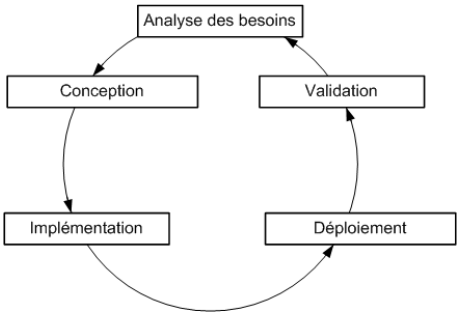


Figure 10.3. *Cycle itératif et incrémental*

Ainsi, le logiciel est développé en effectuant plusieurs itérations chacune apportant un incrément (voir la figure 10.3). Chaque itération correspond à un enchaînement des étapes de développement. Un incrément est un raffinement du logiciel, créé dans l'itération précédente, par ajout de nouvelles fonctionnalités. L'idée générale vise à diminuer la complexité du développement du logiciel en créant d'abord un logiciel avec moins de fonctionnalités et de s'assurer que les fonctionnalités implémentées sont correctes, en effectuant des tests, avant d'ajouter les autres fonctionnalités au fur et à mesure dans les itérations suivantes.

Il existe de nombreux modèles de cycle itératif et incrémental, comme par exemple ceux incorporés dans les nouvelles méthodologies de développement *Extreme Programming* (XP) [BEC 99] et SCRUM [BEE 99]. En 2001, le manifeste agile [5] a été rédigé. Il défini le terme *développement agile*, pour nommer un développement basé sur un cycle itératif et incrémental.

10.3.3. *Les spécificités du développement de logiciels à composants ou orientés services*

Après avoir donné un aperçu de quelques modèles de cycle de développement, nous détaillons les différentes étapes d'un cycle de développement dans le cas des architectures à composants et orientées service, c'est-à-dire les étapes d'analyse des besoins, de conception, d'implémentation, de validation et de déploiement.

10.3.3.1. *Etape d'analyse des besoins*

A l'étape d'analyse des besoins, les analystes définissent le cahier des charges fonctionnel et technique. L'objectif de cette étape vise à déterminer la viabilité et la faisabilité du projet, puis de lister les fonctions du logiciel et ses contraintes.

5. http ://agilemanifesto.org

10.3.3.2. *Etape de conception*

Durant cette étape, l'architecte logiciel décrit globalement l'architecture logicielle sans se soucier des détails d'implémentation. Dans le cas des architectures à composants, la conception de l'architecture consiste à effectuer un raisonnement structurel pour assembler les composants entre eux. Dans le cas des architectures orientées service, la conception correspond à décrire le flot de contrôle, c'est-à-dire l'enchaînement des invocations des services.

Pour faciliter la conception d'architectures, l'architecte dispose de deux approches : les langages de description d'architecture (ADL) [MED 00] et l'ingénierie dirigée par les modèles (IDM) [EST 05]. Ces approches visent à abstraire les concepts requis pour la conception du logiciel afin de faciliter le raisonnement et la construction de l'architecture. Ces approches permettent aussi de vérifier que l'architecture décrite respecte les règles de composition du modèle utilisé. Par exemple, du coté des ADL, il existe Fractal-ADL pour le modèle à composants Fractal, WSDL [W3C 01]/WS-BPEL [OAS 07] pour les services web. Du coté de l'IDM, il existe UML2 [OMG 07b] pour décrire une architecture à composants et BPMN (*Business Process Modeling Notation*) [OMG 07a] pour décrire une architecture orientée service.

10.3.3.3. *Etape d'implémentation*

Durant l'étape d'implémentation, les développeurs codent le logiciel conformément à la spécification écrite par l'architecte. Pour une architecture à composants ou orientée service, l'implémentation équivaut à l'écriture du code des composants primitifs et des services. Cependant, bien que l'architecture logicielle permette un développement modulaire, son implémentation reste difficile à appréhender par les développeurs. En effet, pour chaque composant et service, en plus d'écrire le code métier, c'est-à-dire le code des fonctionnalités offertes par le composant ou le service, les développeurs doivent aussi écrire le code technique spécifique à la plate-forme à composants utilisée ou à la plate-forme d'exécution des services. Ce code permet aux composants et aux services de communiquer entre eux. Or, l'écriture de ce code est laborieux et souvent source d'erreurs.

Dans un souci de faciliter l'implémentation des composants et des services, deux grandes approches existent : la programmation générative [CZA 00] et la programmation par attribut [PAW 05]. Avec la programmation générative, le code technique est généré à partir de l'ADL. La programmation par attribut vise à marquer certains éléments du code pour leur ajouter une sémantique liée aux notions d'architecture logicielle.

10.3.3.4. *Etape de validation*

L'ISO 9126 définit la fiabilité du logiciel comme étant l'aptitude du logiciel à maintenir son niveau de service dans des conditions déterminées pendant une période de temps définie [ISO 03]. Dans le cadre de ce document, nous supposons que les

composants et les services sont conformes à leurs spécifications, c'est-à-dire qu'ils fournissent les fonctionnalités décrites, et nous nous focalisons sur la validation de la compatibilité des interactions entre les composants et les services.

10.3.3.5. *Etape de déploiement*

L'étape de déploiement est réalisée par un administrateur système. Cette étape a pour but de mettre le logiciel à disposition des utilisateurs. Dans le cadre de ce document, nous considérons que les plates-formes d'exécution sont déjà déployées et que, seuls les composants et services constituant le logiciel nécessitent un déploiement. La majorité des plates-formes offre deux méthodes de déploiement : l'ADL et l'API. La méthode de déploiement par l'ADL consiste à déployer la totalité du système à partir de sa description ADL. Elle est effectuée par l'outil de déploiement associé à la plate-forme. La méthode par l'API est une approche programmative. Avec cette méthode, l'administrateur système écrit un script déploiement permettant de déployer et de replier des composants ou des services pendant l'exécution du logiciel.

10.3.4. *Défis de l'évolution du logiciel dans les cycles de développement avec les approches à composants ou orientées service*

Le développement de logiciels à base de composants ou de services repose sur un cycle de développement descendant tel que nous venons de le décrire. De manière générale, les processus de développement ne sont pas adaptés pour suivre l'évolution du logiciel de manière incrémentale et itérative. Pour faire évoluer le logiciel, il serait intéressant que l'architecte puisse le modifier en cohérence avec l'ensemble du cycle de vie, et plus particulièrement les étapes de conception et d'implémentation. De plus, lors de l'exécution du logiciel, la vue conceptuelle requise par l'architecte est perdue et mettre à jour le logiciel en cours d'exécution en utilisant l'ADL n'est possible que si le logiciel est arrêté et redémarré ensuite. En effet, chacune des étapes du cycle de développement reste pour le moment fortement découplée, ce qui peut entraîner l'introduction d'incohérences lors de la transition entre chacune des étapes.

Les défis actuels de l'évolution d'un logiciel dans le cadre d'approches à composants ou orientées services sont de proposer i) un cycle de développement dont les différentes étapes sont fortement couplées et ii) un suivi de l'évolution incrémental et itératif. Pour le couplage fort des étapes du cycle de développement, il s'agira par exemple de proposer un cadre architectural qui soit décrit pendant l'étape de conception et utilisé pendant l'étape de validation et l'étape d'exécution ou encore d'imaginer que les différentes configurations du logiciel décrites pendant la phase d'analyse soient utilisées pour valider le déploiement ou le redéploiement du logiciel. Pour ce qui est du suivi incrémental et itératif, un processus décrivant l'enchaînement des étapes de l'évolution et les dépendances entre ces étapes permettra aux concepteurs de la suivre.

10.4. Fiabilisation des évolutions

Une fois que l'architecte a assemblé les composants et les services correspondant à l'application souhaitée, il a besoin de vérifier si ces composants et services qui interagissent entre eux satisfont les exigences métiers. Puisque l'architecture peut être large et complexe, il n'est pas envisageable qu'il analyse manuellement son architecture pour identifier des violations des exigences. Donc, afin de répondre à ce besoin, quelques modèles offrent à l'architecte un moyen de spécifier ces exigences sous forme de propriétés applicatives ainsi que des outils d'analyse statique de l'architecture qui valident les interactions entre les composants et les services en vue de fiabiliser au mieux le logiciel. Cette section dresse une taximonie des propriétés applicatives et fait un bilan comparatif sur leur prise en compte dans les architectures logicielles.

10.4.1. *Catégories de propriétés applicatives*

Les propriétés applicatives permettent de décrire comment le composant ou le service interagit avec son environnement et comment son environnement peut agir sur lui. Elles expriment les contraintes de réutilisation que le service ou le composant impose sur son environnement, ainsi que les garanties qu'il offre.

Dans ce document, nous affinons la classification des propriétés applicatives en quatre niveaux (basique, comportemental, synchronisation et quantitatif) de Beugnard et al. [BEU 99] et proposons de classer les propriétés applicatives selon les quatre niveaux suivants : structurel, comportemental, de flot de données et de qualité de service (QdS). Nous rappelons dans un premier temps la définition proposée par [BEU 99] et nous déclinons ces propriétés selon notre classement.

Beugnard et al. définissent un modèle de propriété applicative appelé contrat pour la spécification de composants évoluant dans un environnement réparti et concurrent. Leur approche propose quatre niveaux de contrats allant d'une description de propriétés simples comme les contraintes syntaxiques à des propriétés prenant en compte l'environnement, comme les contraintes spatiales et temporelles et celles propres à la qualité de service. Ainsi un contrat d'un composant peut être spécifié par quatre niveaux de contrat :

1) un contrat élémentaire qui comporte des opérations que le composant peut exécuter, les paramètres d'entrées et de sortie qu'il doit avoir et les exceptions qui pourraient survenir lors de l'opération ;

2) un contrat comportemental qui spécifie le comportement des opérations sous forme de pré et postconditions et les invariants ; celui-ci considère les opérations comme des actions atomiques ;

3) un contrat de synchronisation qui décrit les contraintes temporelles (séquence ou parallélisme) entre les différents services proposés ;

4) un contrat de qualité de service qui quantifie le comportement voulu, c'est-à-dire la qualité de service attendue en énumérant les caractéristiques que doit respecter chaque service comme un délai de réponse, un taux de réponse moyenne ou la qualité du résultat.

Nous redécomposons les propriétés associées à ces niveaux de contrats selon la classification suivante :

1) les propriétés structurelles permettent à l'architecte d'exprimer les contraintes structurelles sur l'assemblage formant le logiciel. Ce sont des invariants qui portent sur les composants, les services, les ports et les connecteurs. Ce niveau correspond au premier niveau et à un sous ensemble du second niveau de Beugnard et al. ;

2) les propriétés comportementales autorisent l'architecte à décrire le flot de contrôle entrant et sortant des ports de chaque composant et service. Elles sont similaires au troisième niveau de Beugnard et al. ; Nous avons cependant choisi de renommer le nom de ce niveau dans la mesure où le terme comportement est le plus répandu dans le domaine des architectures logicielles [ALL 97, MAG 99, BAR 09].

3) les propriétés de flot de données permettent de restreindre les valeurs des données entrantes et sortantes des composants et des services. Elles correspondent à un sous ensemble du second niveau de Beugnard et al. ;

4) les propriétés de qualité de services couvrent une large catégorie de propriétés non fonctionnelles incluant la performance, la sécurité et la disponibilité des composants et des services. Elles correspondent au quatrième niveau.

10.4.2. *Prise en compte des propriétés applicatives dans l'évolution des architectures logicielles*

Les travaux autour de SafArchie proposent de classer le résultat d'une analyse d'une interaction selon trois types : compatible, incompatible ou partiellement compatible [BAR 05] :

– L'interaction est dite compatible si les garanties offertes par les composants (respectivement services) remplissent les conditions imposées par les composants (resp. services) assemblés entre eux ;

– L'interaction est dite incompatible quand au moins une des spécifications est violée ;

– L'interaction est dite partiellement compatible quand l'analyse ne déclenche pas d'erreur mais ne peut pas être terminée car au moins une des propriétés dépend de données dont les valeurs ne sont connues que lors de l'exécution du logiciel. Ce cas est typique de l'analyse du flot de données ou de la performance, comme le temps de répons.

10.4.2.1. *Propriétés structurelles*

Il existe de nombreux travaux autour de la validation des propriétés structurelles de l'architecture du logiciel. Parmi tous ces travaux, OCL [OMG 07b] est un langage de contraintes bien connu des architectes logiciels. Il est applicable sur toutes les sortes de modèle. Néanmoins, il n'est pas dédié à la spécification des contraintes sur des modèles à composants ou orientés service. C'est pour cette raison que différents travaux se sont inspirés du langage OCL pour définir des langages de contraintes dédiés aux modèles à composants, comme CCLJ [COL 07] et Acme/Armani [MON 01]. Ces langages ajoutent des constructions de langage pour simplifier l'écriture des contraintes. Tous les outils de vérification associés à ces langages ont besoin de connaître la structure de l'architecture et, dans le cas des outils d'analyse incrémentale, ils requièrent en plus les informations de modification structurelle. Néanmoins, très peu d'approches autorise la vérification incrémentale du logiciel dans le cas d'une évolution de l'architecture. Seuls quelques travaux traitent ce problème, comme [CAB 06].

10.4.2.2. *Propriétés comportementales*

La majorité des langages permettant l'expression de propriétés comportementales reposent sur l'algèbre de processus ou sur les automates, comme Wright [ALL 97], Darwin [MAG 99], BPEL [OAS 07]. Cependant, chacun de ces langages est fortement couplé avec un modèle à composants ou orienté service. En conséquence le choix par l'architecte d'un langage de spécification impose l'utilisation du modèle à composants ou orienté service associé. D'autre part, tous ces travaux ont besoin de connaître la structure de l'architecture et le comportement de chaque composant et service. Dans le cas des services web, le flot de contrôle de l'application, représenté par une description BPMN ou BPEL, est aussi requis. Quant aux analyses incrémentales, elles ont besoin de connaître les modifications structurelles qui se produisent durant l'exécution du logiciel. De plus, afin de supporter les analyses dynamiques du comportement, il est nécessaire de capturer les communications entre les composants et les services qui ont lieu dans la plate-forme. Par exemple, les travaux présentés dans [BAR 08] reposent sur l'observation des échanges de messages entre les services pendant l'exécution du système.

10.4.2.3. *Propriétés de flots de données*

Le support des spécifications de flot de données est très peu répandu dans les modèles à composants ou orientés service. Il repose sur la spécification de pré et de postconditions qui contraignent la valeur des messages échangés, comme dans ConFract [COL 07] et Armani [JUN 07]. Néanmoins, ces approches sont exclusivement dynamiques. Elles nécessitent une observation des données qui sont échangées entre les composants ou les services. A notre connaissance, aucune approche ne propose une analyse statique de flot de données dans les modèles à composants, alors que ce support est très répandu dans le domaine de la validation partielle de programme [KIL 73].

10.4.2.4. *Propriétés de qualité de service*

La gestion des propriétés de QdS est très dépendante du modèle utilisé, comme QuO [PAL 00] et WSLA (*WEB Service Level Agreement*) [IBM 03]. Très peu de modèles fournissent un support des analyses statiques. Généralement ces approches sont issues du domaine des services web (ORBWork [CAR 04]). Elles ont besoin de connaître le flot de contrôle de l'application, c'est-à-dire son comportement, ainsi que les propriétés de QdS de chaque composant ou service. De plus, dans le cas des analyses incrémentales, la capture des informations concernant la modification du comportement est requise. D'autre part, la majorité des approches propose des analyses dynamiques. Ces approches ont besoin de capturer les informations extrafonctionnelles provenant de la plate-forme. Cette capture repose sur l'intégration de sondes dans la plate-forme qui ont pour rôle d'observer les évolutions des valeurs des propriétés de qualité de service.

10.4.3. *Défis de la fiabilisation de l'évolution du logiciel*

Le premier bilan que nous pouvons tirer de cette analyse est, qu'à notre connaissance, il n'existe pas de modèle à composants ou orienté services qui gère les quatre catégories de propriétés applicatives, c'est-à-dire structurelles, comportementales, de flot de données et de QdS telles que nous les avons définies. La majorité des modèles, comme SCA, ne supporte pas la spécification des propriétés applicatives. En général, chaque travail se focalise sur la vérification ou l'analyse d'une seule catégorie de propriétés applicatives. L'exemple le plus remarquable est celui du modèle à composants Fractal. En effet, la plate-forme d'exécution de référence Julia [BRU 04] ne gère aucune propriété applicative. Chaque catégorie de propriétés applicatives est supportée par des extensions différentes de la plate-forme d'exécution, comme par exemple Fractal-BPC [BAR 09] pour les propriétés comportementales et ConFract [COL 07] pour les propriétés de flot de données. Cette multiplication des plates-formes ne permet donc pas à un architecte d'utiliser le modèle Fractal pour vérifier à la fois des propriétés de flot de données et comportementales. En conséquence, le premier défi consiste à proposer un modèle à composants ou orienté service et le cadre outillé associé qui regroupe les quatre propriétés applicatives.

Le deuxième point que nous pouvons extraire de cette comparaison concerne le support des analyses incrémentales. Il existe peu de travaux qui se sont intéressés à l'analyse incrémentale des propriétés. Par exemple, les travaux [CAB 06, BAR 05] et [CAR 04] ont, respectivement, proposé des analyses incrémentales des propriétés structurelles, comportementales et de QdS. Dans les travaux de [TIB 06], on trouve des contrats d'évolution qui permettent de décrire incrémentalement cette évolution et son impact. Les travaux améliorent la rapidité de la vérification. Leur utilisation est donc pertinente pour analyser la fiabilité des évolutions. Le second défi est par conséquent de proposer des analyses incrémentales des propriétés par appui sur le cadre logiciel proposé.

Troisièmement, la très grande majorité des outils d'analyse statique existants ne prend pas en considération les interactions partiellement compatibles. En effet, il n'existe aucun travail qui couple les analyses statiques avec les analyses dynamiques. Les travaux sont, soit purement statiques comme Wright [ALL 97], soit purement dynamiques comme la vérification de flot de données dans Armani [JUN 07]. L'inconvénient majeur de ces approches est un manque de précision dans la prise en compte des interactions partiellement compatibles. Par exemple, les analyses purement statiques peuvent déclarer une architecture fausse car il existe une possibilité de violer une propriété applicative pendant l'exécution, comme le font l'analyse de Wright et de Darwin [MAG 99]. D'autres travaux, comme SafArchie [BAR 05], ne déclarent pas l'architecture fausse mais déclenche une alerte. Les travaux de [TIB 06], par exemple, proposent de lier des descriptions formelles et des attributs de qualité pour permettre des retours aux mainteneurs lors de la détection d'incohérences. C'est donc ensuite à la charge du développeur de mettre en œuvre les analyses dynamiques en ajoutant des tests dans le code du logiciel pour détecter si le problème survient. De même, les travaux purement dynamiques manquent aussi de précision. Ces travaux insèrent des tests dans l'application sans effectuer d'analyse plus globale, comme dans le cas de Armani [JUN 07]. Donc, sans analyse statique, l'architecte n'est informé de la violation de la propriété applicative que durant l'étape de validation dynamique, alors qu'il aurait pu l'être dès l'étape de conception. Globalement, les travaux existants n'utilisent pas la connaissance produite par les outils d'analyse statique pour raffiner les analyses dynamiques. Ce dernier point correspond au troisième défi pour une meilleure fiabilisation du logiciel.

10.5. CALICO : un cadre pour l'évolution fiable de logiciels à composants et orientés services

Les sections précédentes ont mis en évidence le besoin de proposer des cycles de développement prenant en compte l'évolution du logiciel et ceci de manière fiable en mettant en évidence un ensemble de défis dans les sections 10.3.4 et 10.4.3.

Pour traiter cette problématique, cette section présente notre contribution CALICO qui signifie *Component AssembLy Interaction Control framewOrk* [WAI 10]. CALICO est un canevas de développement logiciel itératif et incrémental pour les systèmes à composants et à services (voir figure 10.4). Il permet à l'ensemble des acteurs (architectes, développeurs et testeurs) de concevoir et faire évoluer le logiciel de manière fiable. Pour réaliser cet objectif, CALICO est composé de deux niveaux : un niveau modèle et un niveau plate-forme. Le niveau modèle offre à l'architecte une vue conceptuelle et le niveau plate-forme contient le logiciel en cours d'exécution. Les niveaux modèle et plate-forme sont fortement couplés de telle sorte que l'architecte peut faire évoluer rapidement son logiciel grâce à la vue conceptuelle qui reflète la structure du logiciel. De plus, afin de fiabiliser les évolutions, CALICO associe une analyse statique des évolutions avant leur propagation dans le logiciel avec une

analyse dynamique des évolutions en exécutant le logiciel dans la plate-forme cible. L'architecte logiciel peut donc élaborer et valider son système logiciel à composants ou orienté service en itérant les étapes de conception et les étapes de validation.

10.5.1. *Un cycle de développement itératif et incrémental dirigé par les modèles pour l'évolution*

CALICO offre aux différents acteurs la possibilité d'élaborer et de faire évoluer un système logiciel à composants ou orienté services en effectuant un processus de conception itératif et incrémental. A chaque itération, CALICO permet d'alterner entre les activités de conception et les activités de validation dynamique. Les activités de conception correspondent, d'une part, à la définition de la structure et des propriétés du système par l'architecte logiciel. D'autre part, elles consistent à mettre à jour le système afin de corriger les erreurs identifiées durant les activités de validation dynamique et à faire évoluer le système en fonction du retour des informations remontées par les utilisateurs ou la plate-forme, comme par exemple le temps de réponse. Les activités de validation dynamique consistent à l'exécution des scénarios d'utilisation par les testeurs afin de vérifier dynamiquement la compatibilité des interactions entre les composants et les services. Plus précisément, le cycle de développement itératif et incrémental est composé des huit étapes suivantes (voir la figure 10.4) :

1) conception : L'architecte produit une première conception de son système à l'aide des métamodèles de structure du système et de contrats de CALICO. Ainsi, il conçoit l'architecture de son système et spécifie les diverses propriétés structurelles, comportementales, de flot de données et de QdS que le logiciel doit satisfaire ;

2) analyse statique : L'outil d'analyse des interactions vérifie la cohérence de l'architecture du système, c'est-à-dire qu'il identifie si l'architecture décrite respecte les propriétés applicatives spécifiées. Pour chaque interaction partiellement compatible, il ajoute dans le modèle de débogage l'analyse dynamique qui doit être faite pendant l'exécution du système. Pour chaque interaction incompatible, l'outil notifie l'architecte afin que ce dernier corrige sa conception pour gérer le problème. Tant qu'il reste des interactions incompatibles, les étapes suivantes du cycle ne peuvent pas être effectuées. Cela garantit la production d'un système cohérent, c'est-à-dire qui ne viole pas les propriétés applicatives. De plus, cet outil vérifie aussi les contraintes de la plate-forme cible afin de garantir que l'architecture pourra être déployée sur la plate-forme ;

3) implémentation : pour chaque composant ou service qui n'existe pas, l'outil de génération de code produit le squelette de code. Lors de cette étape, les développeurs implémentent le code métier des nouveaux composants et services ;

4) instrumentation : cette étape a pour but de faire le lien entre les analyses statiques et dynamiques. Pour réaliser cet objectif, l'outil d'instrumentation analyse le modèle de débogage afin d'identifier les analyses dynamiques qui doivent être effectuées. Pour chaque analyse dynamique, l'outil d'instrumentation détermine les données d'exécution nécessaires. Ensuite, l'outil ajoute le support requis pour observer

ces données d'exécution. Pour réaliser cet objectif, l'outil insère les mécanismes d'observation en instrumentant le code de l'application qui est complété par les développeurs et génère des sondes d'observation ;

5) déploiement : l'outil de chargement de l'application instancie le système sur la plate-forme cible conformément à la description de l'architecture faite au niveau modèle. Concrètement, l'outil génère un modèle contenant la succession des opérations élémentaires à effectuer pour instancier chaque élément nécessaire pour former l'application, comme par exemple ajouter ou supprimer un composant. Ensuite, l'outil interprète ce modèle en appelant l'API spécifique à la plate-forme d'exécution pour instancier le système ;

6) exécution : Pendant cette étape, les testeurs exécutent les scénarios d'utilisation du système logiciel, définis pendant l'étape d'analyse des besoins, afin de valider dynamiquement les interactions entre les composants et les services. Ainsi, le système instrumenté réifie les données d'exécution, comme par exemple les changements du contexte d'exécution ou les valeurs des messages échangés, au niveau modèle sous forme d'évènements. Afin de mettre en œuvre automatiquement le mécanisme d'observation des nouvelles données dans la plate-forme, les étapes 4 et 5 sont redéclenchées.

7) validation dynamique : Au niveau modèle, l'outil de débogage analyse les évènements et si besoin, il déclenche l'analyse dynamique appropriée contenue dans le modèle de débogage. Si l'outil identifie une erreur d'exécution, il notifie l'architecte pour que celui-ci corrige sa conception ;

8) évolution de la conception : En accord avec les résultats de la validation dynamique, l'architecte peut modifier sa conception pour corriger le problème. L'architecte peut aussi faire évoluer sa conception afin de l'adapter aux nouveaux besoins des utilisateurs et au changement dans le contexte d'exécution. L'architecte réitère le cycle de développement à partir de l'étape 2 afin d'analyser si les changements dans la conception n'ont pas introduit de nouvelles incohérences.

Dans son ensemble, CALICO est un canevas dédié pour faire évoluer le logiciel. En effet, lors de chaque itération, le système en exécution n'est pas réinstancié entièrement depuis l'état initial. Au contraire, seuls les changements de conception sont propagés dans le système en exécution grâce à l'exécution d'une adaptation dynamique contrôlée par l'outil de chargement. De plus, l'architecte peut aussi modifier sa conception à n'importe quel moment afin de faire face à une évolution inattendue du contexte d'exécution et des besoins applicatifs des utilisateurs.

10.5.2. *Spécification des propriétés applicatives*

Pour concevoir et faire évoluer son logiciel, l'architecte a besoin d'assembler des composants et des services, ainsi que de spécifier les diverses propriétés applicatives

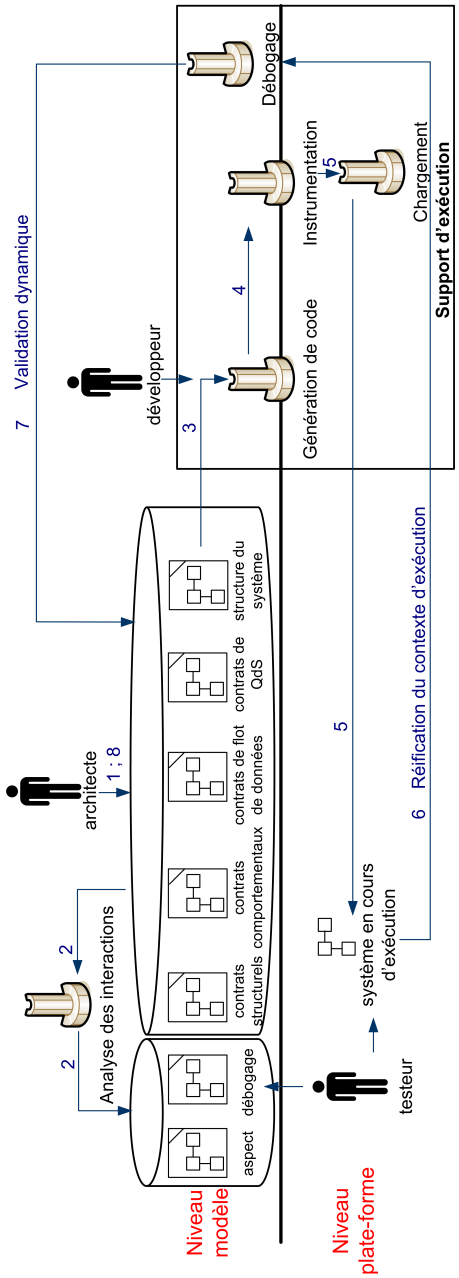


Figure 10.4. *Aperçu de CALICO*

de son logiciel. Dans cet objectif, CALICO offre à l'architecte un ensemble de métamodèles de description d'architecture. Ces métamodèles sont indépendants de tout modèle à composants ou orienté services. La spécification complète d'un système logiciel est exprimée à l'aide de cinq métamodèles (voir figure 10.4). Le métamodèle de structure du système permet la description des composants ou des services ainsi que leur assemblage. Le métamodèle des contrats structurels sert à spécifier les propriétés applicatives structurelles de l'assemblage de composants ou de services. Le métamodèle des contrats comportementaux est utilisé pour décrire le flot de contrôle entrant et sortant des composants ou des services. Le métamodèle des contrats de flot de données permet de définir des hypothèses et des garanties sur les valeurs des messages échangés entre les composants ou les services. Le métamodèle des contrats de qualité de service (QdS) sert à spécifier des propriétés extrafonctionnelles de l'assemblage, comme par exemple des propriétés de sécurité ou liées aux performances.

Afin d'autoriser la conception de la structure d'un système indépendamment de toute plate-forme, nous avons élaboré un métamodèle de la structure du système générique basé sur le métamodèle Ecore. Ce métamodèle résulte d'une analyse de domaine de plusieurs modèles à composants (Fractal, CCM, EJB, OpenCOM) et d'un modèle orienté service (SOA, web service, SCA). Il contient un minimum de concepts et de contraintes de règles de composition communs aux différents modèles. Ces règles portent sur la vérification de la stricte encapsulation des entités et sur l'unicité du nommage des éléments. Les règles de composition spécifiques à chaque plate-forme sont définies séparément du métamodèle grâce aux profils de plate-forme. Cette approche permet de vérifier si une architecture respecte les règles de composition de la plate-forme cible et peut donc en conséquence être déployée sur cette plate-forme. La migration vers une autre plate-forme est ainsi facilitée. L'architecte a besoin de changer le profil chargé et de mettre à jour les propriétés spécifiques à la plate-forme dans la mesure où la spécification de la structure n'est pas dépendante de la plate-forme.

Les spécifications des propriétés applicatives reposent sur le paradigme hypothèse/garantie de Lamport [ABA 93]. De cette manière, CALICO est générique et extensible. En effet, CALICO autorise un expert du domaine à ajouter le support de nouvelles catégories de spécifications. Pour réaliser cette extension, l'expert de domaine a besoin de définir un nouveau métamodèle et de le coupler fortement avec le métamodèle de la structure du système. D'autre part, dans la mesure où le métamodèle de la structure est indépendant de toute plate-forme, la spécification des quatre niveaux des propriétés applicatives est possible pour les différentes plates-formes gérées par CALICO. Par exemple, un architecte peut spécifier les quatre niveaux de propriétés applicatives sur une architecture SCA, alors que nativement les plates-formes d'exécution de ce type ne le permettent pas. Globalement, le mécanisme mis en place pour l'étape de conception permet une réunification générique des travaux existants dans le domaine de la conception. De plus, ces travaux qui étaient auparavant dédiés à une plate-forme sont maintenant rendus disponibles pour toutes les plates-formes prises en compte dans CALICO.

10.5.3. *Couplage des analyses statiques et dynamiques*

Afin de s'assurer que la description de l'architecture est correcte et qu'aucune propriété applicative n'est violée, la cohérence de l'architecture est vérifiée statiquement par l'outil d'analyse des interactions entre les composants et les services. L'analyse détermine toutes les interactions entre les composants et les services à l'aide du métamodèle de la structure du système. Pour chaque interaction, elle analyse les propriétés applicatives des composants et des services participant à l'interaction, qui sont spécifiées dans les quatre métamodèles de contrats et en déduit la catégorie de l'interaction. CALICO prend en compte les trois catégories d'interaction : compatible, incompatible et partiellement compatible (voir section 10.4.2).

Dans le but d'obtenir une architecture cohérente, pour chaque interaction incompatible, l'outil notifie le problème à l'architecte afin que celui-ci corrige la conception de son architecture. Pour chaque interaction partiellement compatible, l'outil dispose de deux métamodèles, le métamodèle de débogage et le métamodèle d'aspect, qui permettent de définir le lien avec l'étape de validation dynamique, de manière générique. Le métamodèle de débogage contient la description de l'ensemble des analyses dynamiques qui doivent être effectuées pendant l'exécution du système. Le métamodèle d'aspect repose sur le principe de programmation orientée aspect et il sert à spécifier des fonctionnalités transverses requises pour observer les données d'exécution nécessaires aux analyses dynamiques. De cette manière, l'outil d'analyse statique peut insérer dans le modèle de débogage les analyses dynamiques qui doivent être effectuées pendant l'exécution du logiciel afin de finaliser la validation des interactions partiellement compatibles.

La mise en œuvre des analyses dynamiques est entièrement prise en charge par CALICO. En effet, CALICO analyse le modèle de débogage afin d'ajouter et de supprimer des sondes du système pour ne capturer exclusivement que les événements requis pour finaliser la validation des interactions partiellement compatibles. Cela évite un surcoût lié à l'observation d'événements inutiles. Les erreurs sont directement remontées au niveau modèle et sont compréhensibles par l'architecte logiciel. Ainsi, l'architecte peut, à n'importe quel moment, ajuster sa conception pour corriger le problème dans l'itération suivante du cycle de développement.

Globalement, CALICO permet d'effectuer une validation dynamique d'un système même si la plate-forme d'exécution sous jacente ne fournit aucun mécanisme d'analyse dynamique. La plate-forme a uniquement besoin de gérer les adaptations dynamiques, ce qui est une fonctionnalité courante des plates-formes récentes.

10.5.4. *Implémentation et évaluation*

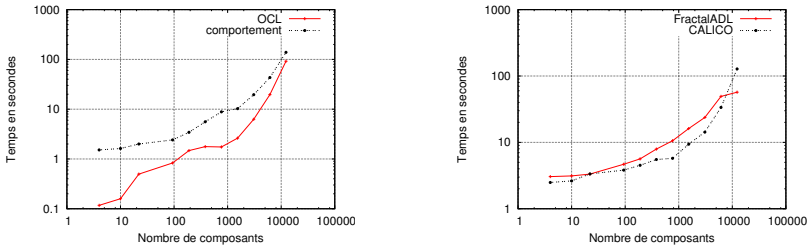
Afin d'évaluer nos travaux, nous avons élaboré une implémentation conséquente de CALICO. Notre implémentation de CALICO est complètement intégrée dans l'IDE

Eclipse. Elle prend en charge cinq plates-formes (Fractal, OpenCCM, OpenCOM, FraSCAti et GlassFish) et est capable d'effectuer les quatre catégories d'analyse. Elle offre aussi des vues graphiques permettant à un architecte de concevoir son système. Un travail important a été effectué pour permettre l'extensibilité. Il permet l'ajout de nouvelles plates-formes, analyses ou sondes pendant son exécution et donc sans interrompre l'architecte logiciel dans son travail de conception et de validation dynamique.

Nous avons évalué les performances de chacun des outils de CALICO. Tous les tests ont été effectués sur un portable doté d'un processeur Intel®Core™2 Duo cadencé à 1,33 GHz. La machine virtuelle Java de SUN®version 1.6.0_11 a été utilisée. Les expérimentations ont été effectuées sur la plate-forme Fractal.

Nous avons mesuré le temps mis par l'outil d'analyse des interactions. La partie a) de la figure 10.5 représente le temps mis en secondes pour analyser les propriétés structurelles et comportementales du système en fonction du nombre de composants. L'analyse structurelle est très rapide, elle dure moins de 2,5 secondes pour analyser un système de 1500 composants. L'analyse du comportement est, quant à elle, beaucoup plus longue. Elle dure environ 10 secondes pour un système de 1 500 composants. Donc, afin de ne pas contraindre l'architecte à attendre longtemps le résultat de la validation dynamique de son système, nous lui permettons de sélectionner les analyses qu'il désire appliquer sur son système.

Ensuite, nous avons évalué le surcoût de CALICO pour déployer le système. Ainsi, nous avons comparé le temps mis pour déployer le système avec l'outil de chargement de CALICO, par rapport au temps mis avec l'outil de déploiement natif Fractal ADL (voir partie b) de la figure 10.5). Les tests mettent en évidence que CALICO est aussi performant que l'outil natif pour déployer un système. Par exemple, pour déployer un système de 3 000 composants, CALICO met quatorze secondes alors que Fractal ADL met vingt-trois secondes. Nous avons aussi mesuré le temps mis pour effectuer un déploiement incrémental, c'est-à-dire le temps mis pour ajouter un nouveau composant dans un système déjà déployé. Jusqu'à 3 000 composants, la complexité est linéaire,



a) *Durée d'analyse* **b)** *Durée du déploiement*

Figure 10.5. *Evaluation de CALICO*

CALICO met environ six secondes pour effectuer la mise à jour. Le temps obtenu est donc raisonnable pour une activité de validation dynamique du système.

Globalement, les tests de performances que nous avons réalisés sur notre implémentation mettent en évidence le fait que CALICO offre des performances suffisantes pour faire évoluer des applications jusqu'à 10 000 composants et services, ce qui correspond à la montée en charge maximale de nombreuses plates-formes existantes.

10.6. Conclusion

Les systèmes logiciels modernes se distinguent par un besoin d'évolutions rapides. Ces évolutions sont nécessaires pour prendre en compte les nouvelles exigences des utilisateurs. De plus, un logiciel a besoin de constamment satisfaire les contraintes de fiabilité et de qualité de service exigées. Or, la fiabilisation des évolutions reste un défi. En effet, les processus de développement des architectures logicielles ne sont pas adaptés pour faire évoluer un logiciel rapidement et de manière fiable. D'une part, des incohérences peuvent être introduites entre chacune des étapes du développement qui sont faiblement couplées. D'autre part, il ne permet pas d'analyser la cohérence des évolutions. A notre connaissance, aucune architecture logicielle autorise la spécification et la vérification des quatre catégories de propriétés applicatives, c'est-à-dire structurelle, comportementale, de flot de données et de QdS. Il y a un manque d'intégration des outils d'analyses statiques et dynamiques.

Afin de répondre à cette problématique, nous avons proposé CALICO, un canevas de développement pour l'évolution fiable de logiciels à composants et orientés services. Ce support repose, à la fois, sur un couplage fort de la vue conceptuelle de l'architecture avec le logiciel en cours d'exécution, et sur l'utilisation d'un cycle de développement itératif et incrémental. CALICO offre à l'architecte une large palette de spécification couvrant les quatre catégories de propriétés applicatives. De cette manière, lors de chaque évolution, CALICO valide la compatibilité des interactions entre les composants et les services, et vérifie si les exigences applicatives sont toujours respectées. Concrètement, nous réalisons cet objectif, non pas en proposant de nouvelles analyses, mais en offrant un cadre fédérateur qui autorise la réutilisation de la plupart des outils d'analyse statique pour les architectures logicielles et des outils d'analyse dynamique du logiciel qui sont dispersés dans les différentes plates-formes existantes. De plus, CALICO est indépendant de toute plate-forme d'exécution. Il permet de résoudre le problème lié au manque de fonctionnalité de validation des évolutions dans les plates-formes d'exécution grâce à un mécanisme d'instrumentation. Ainsi, il offre un support pour la validation statique et dynamique des évolutions à des plates-formes qui n'en disposent pas nativement.

Pour conclure ce chapitre, nous pouvons affirmer que l'enjeu majeur futur concerne la validation des évolutions dans les applications autonomes tels qu'esquissée dans les

travaux de Oreizy [ORE 98]. En effet, ces applications sont capables de se reconfigurer automatiquement, pendant leur exécution, en fonction des changements provenant de leur environnement ou des besoins utilisateurs. Néanmoins, ces évolutions peuvent violer les propriétés applicatives du logiciel. La piste des analyses incrémentales de propriétés applicatives semble prometteur, car ce côté incrémental permettra de minimiser le surcoût lié au temps d'analyse.

10.7. Bibliographie

[ABA 93] ABADI M., LAMPORT L., « Composing specifications », *ACM*, vol. 15, 1993.

[ALL 97] ALLEN R., A Formal Approach to Software Architecture, PhD thesis, Carnegie Mellon, School of Computer Science, CMU-CS-97-144, janvier 1997.

[BAR 05] BARAIS O., Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants, Thèse, Laboratoire d'Informatique Fondamentale de Lille, 2005.

[BAR 08] BARESI L., GUINEA S., PASQUALE L., « Towards a unified framework for the monitoring and recovery of BPEL processes », *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB'08)*, 2008.

[BAR 09] BARROS T., AMEUR-BOULIFA R., CANSADO A., HENRIO L., MADELAINE E., « Behavioural models for distributed Fractal components », *Annals of Telecommunications*, vol. 64, n°1/2, p. 25-43, Springer, février 2009.

[BEA 07] BEA, IBM, INTERFACE21, IONA, ORACLE, SAP, SIEBEL, SYBASE, Assembly Component Architecture - Assembly Model Specification Version 1.00, mars 2007.

[BEC 99] BECK K., ANDRES C.*Extreme Programming Explained : Embrace Change (2nd Edition)*, Addison-Wesley, 2004.

[BEE 99] BEEDLE M., DEVOS M., SHARON Y., SCHWABER K., SUTHERLAND J., « SCRUM : An extension pattern language for hyperproductive software development », *Pattern Languages of Program Design*, vol. 4, p. 637-651, 1999.

[BEU 99] BEUGNARD A., JÉZÉQUEL J.-M., PLOUZEAU N., WATKINS D., « Making Components Contract Aware », *Computer*, vol. 32, n°7, p. 38-45, IEEE Computer Society Press, juillet 1999.

[BRE 07] BREIVOLD H. P., LARSSON M., « Component-Based and Service-Oriented Software Engineering : Key Concepts and Principles », *Proceedings of the 33rd Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, Washington, Etats-Unis, 2007.

[BRU 04] BRUNETON E., COUPAYE T., LECLERCQ M., QUÉMA V., STEFANI J.-B., « An Open Component Model and Its Support in Java », *Proceedings of the 7th International Symposium Component-Based Software Engineering*, Edimbourg, UK, 2004.

[CAB 06] CABOT J., TENIENTE E., « Incremental Evaluation of OCL Constraints », *18th International Conference on Advanced Information Systems Engineering*, Luxembourg, Luxembourg, 2006.

[CAR 04] CARDOSO J., SHETH A., MILLER J., ARNOLD J., KOCHUT K., « Quality of service for workflows and web service processes », *Web Semantics : Science, Services and Agents on the World Wide Web*, vol. 3, n°1, p. 281-308, avril 2004.

[COL 07] COLLET P., MALENFANT J., OZANNE A., RIVIERRE N., « Composite Contract Enforcement in Hierarchical Component Systems », *Proceedings of the 6th International Symposium (SC 2007)*, vol. 4829, Springer-Verlag, p. 18-33, mars 2007.

[CZA 00] CZARNECKI K., EISENECKER U., *Generative programming : methods, tools, and applications*, ACM Press/Addison-Wesley, New York, Etats-Unis, 2000.

[ERL 05] ERL T., *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*, Prentice Hall, Upper Saddle River, Etats-Unis, 2004.

[EST 05] ESTUBLIER J., VEGA G., IONITA A. D., « Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications », *Model Driven Engineering Languages and Systems (MODELS'05)*, vol. 3713, Springer-Verlag, p. 69-83, octobre 2005.

[FAB 07] FABRESSE L., Du découplage à l'assemblage non-anticipé de composants (Conception et mise en œuvre du langage à composants SCL), Thèse, Université Montpellier II, Montpellier, 2007.

[IAB 97] IABG, The Development Standards for IT Systems of the Federal Republic of Germany, The V-Model, http ://v-modell.iabg.de., juin 1997.

[IBM 03] IBM, WSLA Language Specification, V1.0, 2003.

[IEE 00] IEEE 2000, EEE Product No. : SH94869-TBR : Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard No. 1471, 2000.

[ISO 95] ISO/IEC, ISO/IEC 12207, Systems and software engineering – Software life cycle processes, août 1995.

[ISO 03] ISO/IEC, ISO/IEC 9126, Software engineering – Product quality, juillet 2003.

[JUN 07] JUNG H., RUBIO-MEDRANO C. E., WONG W. E., CHEON Y., Architectural Assertions : Checking Architectural Constraints at Run-Time, mai 2007.

[KIL 73] KILDALL G., « A Unified Approach to Global Program Optimization », *1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Boston, Etats-Unis, 1973.

[LAR 03] LARMAN C., BASILI V., « Iterative and Incremental Development : A brief History », *IEEE Computer Society*, vol. 36, n°6, p. 47-56, juin 2003.

[LEH 85] LEHMAN M. M., BELADY L. A., *Program Evolution : Processes of Software Change*, Academic Press, San Diego, Etats-Unis, 1985.

[LOW 05] LOWY J., *Programming .NET Components, 2nd édition*, O'Reilly, 2005.

[MAG 99] MAGEE J., « Behavioral analysis of software architecture using ltsa », *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society, p. 634-637, 1999.

[MED 00] MEDVIDOVIC N., TAYLOR R. N., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n°1, p. 70-93, janvier 2000.

[MON 01] MONROE R., Capturing Software Architecture Design Expertise with Armani, Report, Carnegie Mellon university, Pittsburgh, Etats-Unis, janvier 2001.

[OAS 07] OASIS, Web Services Business Process Execution Language v2.0, avril 2007.

[OAS 08] OASIS, Reference Architecture for Service Oriented Architecture v1.0, avril 2008.

[OMG 07a] OMG, Business Process Model and Notation (BPMN) 2.0, juin 2007.

[OMG 07b] OMG, Unified Modeling Language (UML) : Superstructure, v2.1.1, août 2007.

[ORE 98] OREIZY P., MEDVIDOVIC N., TAYLOR R. N., « Architecture-based runtime software evolution », *Proceedings of the 20th international conference on Software engineering*, ICSE '98, Washington, DC, Etats-Unis, IEEE Computer Society, p. 177–186, 1998.

[OSG 07] OSGI ALLIANCE, OSGi Service Platform Core Specification v4.1, avril 2007.

[PAL 00] PAL P., LOYALL J., SCHANTZ R., ZINKY J., SHAPIRO R., MEGQUIER J., « Using QDL to specify QoS aware distributed (QuO) application configuration », *Proceedings of Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, Newport Beach, Etats-Unis, mars 2000.

[PAW 05] PAWLAK R., « Spoon : Annotation-Driven Program Transformation - The AOP Case », *Proceedings of the 1st Middleware Workshop on Aspect-Oriented Middleware Development*, ACM, p. 1-6, New York, Etats-Unis, novembre 2005.

[ROY 87] ROYCE W., « Managing the development of large software systems : concepts and techniques », *Proceedings of the 9th international conference on Software Engineering)*, p. 328-338, Monterey, Etats-Unis, 1987.

[Sun 97] Sun Microsystems, Enterprise Java Beans, `www.javasoft.com/products/ejb`, 1997.

[SZY 97] SZYPERSKI C., *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley Professional, Boston, Etats-Unis, 1997.

[TIB 06] TIBERMACINE C., FLEURQUIN R., SADOU S., « On-Demand Quality-Oriented Assistance in Component-Based Software Evolution », *CBSE*, p. 294-309, 2006.

[W3C] W3C, « www.w3.org/TR/2004/NOTE-ws-gloss-20040211/, 2004 ».

[W3C 01] W3C, Web Services Description Language (WSDL) 1.1, mars 2001.

[WAI 07] WAIGNIER G., LE MEUR A.-F., DUCHIEN L., « FIESTA : A Generic Framework for Integrating New Functionalities into Software Architectures », *Proceedings of 1st European Conference on Software Architecture (ECSA'07)*, Madrid, Espagne, 2007.

[WAI 10] WAIGNIER G., Canevas de développement agile pour l'évolution fiable de systèmes logiciels à composants et orientés services, Thèse, Université des Sciences et Technologie de Lille, 2010.